# Extracting Temporal Properties from Real-Time Systems by Automatic Tracing Analysis

Andrés Terrasa*[1] and Guillem Bernat[2]

[1] Departamento de Sistemas Informáticos y Computación
Technical University of Valencia (SPAIN)
aterrasa@dsic.upv.es

[2] Real-Time Systems Research Group
Department of Computer Science
University of York (UK)
bernat@cs.york.ac.uk

**Abstract.** Statically analyzing real-time systems normally involves a high degree of pessimism, but it is necessary in systems requiring 100% guarantee. However, lots of less critical systems would significantly benefit from combining such static analysis with empirical tests. Empirical tests are based on observing the system at run time and extracting information about its temporal behavior. In this sense, this paper presents a generic and extensible framework that permits the extraction of temporal properties of real-time systems by analyzing their run-time traces. The analysis is based on event-recognition finite state machines that compute the temporal properties with a computational cost of $O(1)$ per observed event in most of the cases. The framework is instantiated in order to extract some typical temporal properties (such as computation time or response time of tasks), which can serve as a template to define new ones. Finally, the paper also shows how the framework can be implemented on a real system, exclusively using state-of-the-art technology; in particular, the Trace and Real-Time Extensions of the POSIX standard.

## 1 Introduction

One of the main problems of statically analyzing real-time systems is that it usually involves a high degree of pessimism. The typical a priori, *analytical* study of the system checks the worst-case scenario for each task against the task requirements. However, this worst-case scenario, which can make the analysis reject a task, may not actually happen at run time. For example, consider the highest priority task in a system where resources are shared by using some sort of Priority Ceiling Protocol. The worst-case blocking time of this task is the longest critical section of all the low priority tasks sharing a resource with the task. If this blocking time is too large for the task temporal requirements, then the

* This work was developed during a research stay of this author at the University of York, and partially funded by grant CTIDIB/2002/61 from the Valencian Government (Spain) and by EU funded project FIRST (IST-2001-34140).

task set will be rejected by the off-line analysis, even though the lower priority task may *never* happen to block the highest priority task (for example, if their periods are harmonic). In fact, the pessimism is likely to be higher, considering that techniques for calculating the critical section worst-case execution time have probably overestimated it.

Despite its inherent pessimism, this analytical study is required in systems needing 100% guarantee (such as mission-critical systems). However, lots of other less critical systems would significantly benefit from *combining* such static analysis with *empirical* tests. Empirical tests are based on *observing the system at run time and extracting information about its temporal behavior*. This benefit will be shown with three examples featuring different types of systems. In the first example, consider a hard system on which the off-line analysis has rejected a task because its worst-case response time slightly exceeds its deadline. If empirical tests show that the analytical test was very pessimistic, then the system designer may decide to run the system anyway, knowing that the probability of this worst-case scenario actually happening for the task is very low. The second example is a system with a mixed set of hard and soft tasks, scheduled by some sort of flexible scheduling algorithm in order to improve the overall response quality. Flexible scheduling techniques are speculative, in the sense that they try to guess which is the most effective way to schedule tasks in order to maximize the obtained value. If their speculations are based on pessimistic values, then part of their effectiveness is lost; conversely, more accurate, actually observed values can improve the overall scheduling process. The third and final example would be a soft real-time system, on which no schedulability analysis is normally performed. Again, this system would benefit from having a consistent mechanism of testing the system performance, allowing the designer to tune it appropriately. Please note that our aim is not to substitute the off-line analysis by empirical tests on systems requiring hard guarantees, but to *combine* the two approaches order to obtain the best of both.

Empirical tests involve two different activities: extracting run-time information and analyzing it. The extraction of system information at run-time can be done by several different *instrumentation* techniques, ranging from pure hardware to pure software mechanisms, with mixed hardware-software alternatives in the middle. In this range of solutions, there is a double (inverse) gradation of intrusiveness and price, with pure hardware instrumentation being at one end (least intrusive and most expensive) and pure software instrumentation being at the other end.

The analysis of the run-time information, which is usually called execution *trace*, can be performed by following a custom-made approach or by applying a general framework. In this paper, we introduce a general framework for extracting temporal properties from real-time systems by analyzing run-time traces. The characteristics of this framework are summarized as follows:

- The system needs to be instrumented, but the actual instrumentation technique is not imposed by the framework. The instrumentation typically places trace points at both the operating system and the application code.

- As the system runs, trace points generate events, which are sequentially stored to form a run-time trace.
- Traces are automatically analyzed in order to reconstruct the system evolution as state transitions.
- Temporal properties are defined as function over pre-defined sequences of state transitions. These sequences are detected by using finite state machines over the observed trace.
- The maximum computational cost of processing each observed event is $O(n)$, with $n$ being the number of tasks, while the regular cost of many of the events is $O(1)$. This feature limits the properties that can be studied by the framework, but ensures the predictability and bounds the overhead related to the analysis. This, in turn, permits performing on-line analysis of events, if necessary.

The framework defined here is generic and extensible, meaning that it is intended to capture any interesting aspect of the system temporal behavior. However, since one of the main benefits of empirical tests is to compare their results with off-line analyses, the natural properties to be studied are those that are calculated by these analyses. These include system-related properties, such as utilization, and task-related properties such as response times, computation times, blocking times, jitter factors, etc. The paper presents how to instantiate the framework in order to deduce some of these properties, which can serve as a guideline on how to define new ones.

Finally, the paper also introduces a case study of the framework, entirely based on the POSIX suite of standards. In particular, the extraction of the properties described on the paper has been implemented on RT-Linux, a real-time kernel that follows a subset of the POSIX real-time extensions called the Minimal Realtime System Profile [P1003.13]. This kernel has been enhanced by a trace subsystem, conforming with the POSIX Trace standard [P1003.1], which has been used as the actual mechanism of extracting the traces [Ter01]. The conclusion of the case study is that POSIX-conforming systems can successfully adopt the framework in real-life applications; and this, in turn, proves the framework's ability to be employed in systems using state-of-the-art, standard technology.

The paper outline is as follows: Section 2 presents the related work, including a brief survey of approaches using trace analysis and a summary of the POSIX Trace standard and the RT-Linux operating system. Section 3 introduces the general framework for automatically extracting temporal properties. Section 4 details how the framework can be used for study some of the most interesting temporal properties of a real-time system. This section is intended to be as a guide to any new temporal property that the designer is interested in. Section 5 presents the case study, on which the framework has been implemented in a POSIX-like real-time operating system. Implementation issues and results from the property extraction are also presented. Finally, Section 6 states the conclusions of the paper and proposes some future lines of work.

## 2 Related Work

### 2.1 Trace Analysis

The approach of studying the behavior of a system by observing its events is not novel, although it has been mainly applied to other, non-real-time domains. This section summarizes some contributions in this area.

Probably, one of the first approaches of expressing the behavior of a system in terms of traces, events, states (processes) and transitions is the theory of Communicating Sequential Processes (CSP) [Bro84]. The purpose of such theory is to define a mathematical space of processes (along with its operators), by which it is possible to formally demonstrate communication properties of such processes.

Event Based Behavior Abstraction (EBBA) [Bat95] is a general model intended to be used as a debugging formalism in distributed systems. It is based on a hierarchy of events, on which the lowest (primitive) event are directly observed from the distributed system under study, while higher level events are based on clustering of primitive or other higher level events. In short, EBBA provides an event recognition engine and an behavioral analysis model provided by the user (based on the EDL grammar). The comparison between the events detected by the engine and the models provided by the user permits to point out whether the system behavior is correct or not.

FORMAN (FORMal ANnotation) language [Aug95] is a general framework for debugging programs. It is based on two concepts: an event grammar that permits an automatic instrumentation of the program source code (implemented in a high-level *target* language), and a language to express computations over the program trace (or *H-space*), after its execution. The language is used to write assertions over single events or event patterns (sequences) on the trace, allowing for the evaluation of debugging rules, queries of variable values, profile information, statistics, etc.

The trace assertion method for abstract specification of software [Bar77,McL84] is a formal methodology for program specification on which programs are abstractly specified by means of its observable features or properties (outputs as a function of inputs), rather than by the algorithms that they implement. The specification method is founded on describing (1) which functions of the program can be called (events), (2) which are the legal sequences of these calls (traces) and (3) which is the observable output at the state reached by each legal sequence. The methodology was initially intended for demonstrating formal properties of the program specifications, such as consistency and completeness, but later work has used it for other purposes, such as specifying (and reproducing) program behaviors in terms of assertion about traces [Wan93].

Our framework shares, or inherits, some of the key ideas of such pieces of work, such as deducting or validating behaviors by observing the system evolution, although with a complete different aim. The main purpose in our approach is not to specify, demonstrate or debug *functional* properties of a system, but to deduce *temporal* ones. The issue of temporal behavior of real-time systems has been actually addressed by some formalisms such as the timed automata [Alur94]; however, these formalisms are normally focused on validating the intended behavior of a system, rather than extracting actual temporal properties.

Finally, another main difference between the framework proposed here and most of previous work, with maybe the exception of the work by Stewart et al. [Ste92], is that the main source of event information is not the program to be run, but the operating system. Therefore, instrumentation is primarily done at the operating system code. Depending on the operating system support and on the properties that the user is interested on, it may be necessary to instrument the program source code too, but only as a complement to the information obtained from the operating system.

### 2.2 The POSIX Trace Standard

The POSIX Trace standard establishes a set of portable interfaces which allow applications to have access to trace management services, which are implemented at the operating system level. This standard has been recently integrated within the last approved version of the complete POSIX suite of standards [P1003.1]. The POSIX Trace standard is founded on two main data types (trace event and trace stream) and is also based on three different roles which are played during the tracing activity (the trace controller process, the traced process and the analyzer process). These concepts are now detailed.

A *trace event* is defined as a data object representing an action which is executed by either a running process (user trace event) or by the operating system (system trace event). User events are explicitly generated by the application, by calling a specific trace function, while system events are internally generated by the operating system. Each user or system event belongs to a particular *trace event type* (an internal identifier) and it is associated with a *trace event name* (a human-readable string), by which it can be later recognized. Each time an event is traced, the trace system is required to store some information about the event including, among others, the identifier of the process (and thread) that traced the event, the memory address at which the trace function was called and a timestamp (with a minimum precision of one microsecond). Optionally, arbitrary extra data can be associated with the event when the application or the operating system trace it. All this information, mandatory and optional, has to be reported when the event is retrieved for analysis.

A *trace stream* is defined as a non-persistent, internal (opaque) data object containing a sequence of trace events plus some internal information to interpret those trace events. That is, streams are where events are stored (when traced) and where events are read from (when retrieved). The standard defines that a trace stream is explicitly created to trace a particular process (or a set of related process). Trace events can be either retrieved directly from the stream (on-line analysis) or permanently stored in a file called a log. In the latter case, analysis can only be done after the tracing is over and all the events have been stored in the log file (off-line analysis).

The standard defines three different roles in each tracing activity: trace controller process (TCP), traced (or target) process (TP) and trace analyzer process (TAP). In the most general case, each of these roles is executed by a separate process. However, nothing in the standard prevents from having two (or even the three) of these roles executed by the same process (possibly by different threads). The TCP is the process that sets the tracing system up, including the creation of the stream(s), in order to trace a (target) process. The TP is the process that is being traced. This process can only register new user event types and trace events belonging to these types. The TAP is the process in charge of retrieving the stored events from a trace stream, in order to analyze them. If the stream does not have a log, then both the target and the analyzer processes access the stream concurrently, the former for tracing events and the latter for retrieving these events.

### 2.3 RT-Linux and its Implemented POSIX Standards

RT-Linux (Real-Time Linux) [Yod99] is a small real-time operating system which is able to concurrently execute a set of real-time threads and a Linux *system*, which here refers to the Linux kernel plus all the user processes being executed by the kernel. The sharing of the processor between the real-time tasks and the Linux system is done in a complete unfair way, in the sense that real-time tasks are always given the processor in preference of the Linux system and without delays, independently of the actions taken by the Linux kernel. These bounded latencies are achieved by giving the Linux kernel a set of *virtual* interrupts instead of the real (hardware) ones, which are managed by RT-Linux only. Typical RT-Linux applications are decoupled in two parts: a real-time part, comprising a set of real-time tasks, and a non-real-time part, implemented as one or more Linux processes. Both parts can communicate via either shared memory or bidirectional channels called RT-fifos.

RT-Linux has lately adopted the external interface established by the POSIX Minimal Realtime System profile [P1003.13], which describes the requirements of the smallest POSIX-conforming real-time operating system. On the hardware side, these requirements include only one processor, no explicit memory protection, no mass storage devices and, in general, simple hardware devices operated synchronously. On the software side, the requirements establish a simple programming model in which the real-time system is executed by only one process (with complete POSIX thread support [P1003.1]), without the need of a file system or user interaction.

The POSIX Trace standard has been recently added to RT-Linux (see [Ter01] for a detailed description). The implementation has restricted the subset of the standard to the case of having on-line analysis of events only, but with the possibility of dynamically filtering of event types. Following the RT-Linux model, the trace support has made available at both the RT-Linux and Linux levels. As a result, events can be traced and retrieved by real-time tasks and Linux processes *concurrently*.

## 3 Systems, Events, States and Properties

This section introduces the framework to deduce temporal properties by observing the real-time system at run time. In short, the framework is based on four concepts: the *system* to be studied, the set of *events* that can be observed from the system, the set of *states* that can be derived from the events and the *properties* that can be calculated through the observation process. The framework's general structure is pictured in Figure 1, while each of these concepts is detailed below.

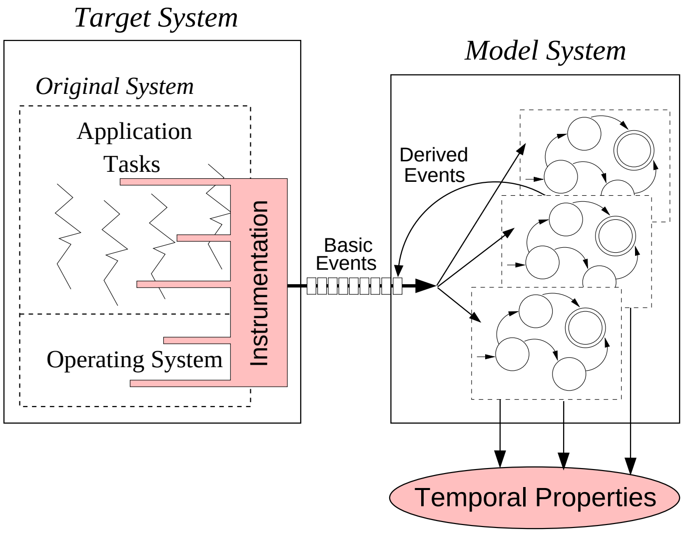



**Fig. 1.** An outline of the framework.

The framework distinguishes among three different systems: the original system, the target system and the model system. The *original system* ($S$) is the real system under study, before being instrumented. This system is formed by a set of runnable real-time tasks (potentially belonging to different real-time applications) plus an operating system or kernel in charge of scheduling and running these application tasks. The real system cannot be observed unless some instrumentation is performed over it. Since the framework does not impose a particular instrumentation mechanism, it cannot assume that the instrumentation does not affect the behavior of the original system. Therefore, the instrumented system is actually our *target system* ($T$), the system which properties are deduced from. The *model system* ($M$) is the view of the target system that can be deduced by means of the observation process, that is, it is the model of the target system. The model system is defined in terms of a set of properties of interest plus a set of finite state machines, which are used as recognition engines for events and as computation tools for properties. In general, if model system $M$ successfully represents the behavior of the target system $T$, the properties than can be deduced in $M$ also hold in $T$. These properties might also be extrapolated to $S$, probably with just some loss of precision. However, the exact influence of the instrumentation in the original system's behavior heavily depends on the actual instrumentation technique, and hence such extrapolations should be done carefully.

As the target system executes, its internal state changes. Some of these state changes are detected by the instrumentation and converted into basic events, which are the observation individuals. When observed, these basic events trigger state transitions in the model system. Some of these transitions can generate other (derived) events, and so on. The framework defines properties as functions over particular sequences of state transitions, with these transitions being detected by finite state machines (deterministic finite automata).

An *event* is a value that triggers an instantaneous state transition in the model system. In fact, since the system model is likely to be internally formed by several automata, a single event may produce transitions in more than one automaton (although this may still be seen as a single state change for the entire system model). For each automata, a transition is triggered if the last produced event matches the *transition condition*, which is formed by an event pattern and some logical expressions. There are two types of events: basic and derived. *Basic events* are directly observed from the target system, that is, they are produced by the instrumentation at either the operating system or the application code. These events usually represent significant state changes in the target system. *Derived events* are triggered inside the model system, when a particular sequence of events (basic or derived) is detected. The model system uses the same recognition engine, based on finite state machines, to trigger derived events and to calculate system properties. The process works like this: a basic event normally triggers a state transition in one or more automata; the result of these state transitions may result in a derived event being produced, which can in turn produce state transitions in the same or other automata. Note that, if the event recognition process is to be done on-line, then there must be an upper bound in the amount of events that can be successively derived by a single basic event being observed, in order to bound the computation time of the property extraction process. Each event is defined by means of an event type, a timestamp (which indicates the moment at which the event was either observed or derived) and, optionally, some extra data depending on the event type (as, for example, the task identifier in a task-related event type). The purpose of the timestamp is to partially order events[3].

At any particular point in time, the target system and the model system are each in a certain *state*. Strictly speaking, the state of the target system is the current hardware state, plus the internal state of the operating system, plus the internal state of each process (and each task inside the process). However, the

[3] Although this is not imposed by the framework, typical temporal properties of a real-time system will need the timestamp of basic events to be set by an absolute, monotonic real-time clock in the target system.

framework assumes that there is only a (normally small) subset of this state that can be detected by the instrumentation. This is called the set of *observable states* of the target system. State changes between observable states are notified by means of basic events. In the model system, the state of the system is the current state of all the automata used to derive events and properties plus the current value of all the properties. The calculation of some properties may need the use of auxiliary variables; if so, the current value of all the auxiliary variables also forms part of the model system state.

A *property* is some value about the system execution which is of particular interest. Each property is abstractly defined as a function of a sequence of states (or transitions). Conceptually, the model system defines a different finite state machine or automaton to calculate each property, although the implementation may join some of these automata in order to improve performance. The calculation of each property is actually carried out by means of *semantic rules*, which are pieces of code that can be attached to transitions and states of the automaton; these semantic rules are executed when the transition is triggered or when the state is reached. As a result, the evolution of the automaton naturally computes the property, whose final value is calculated when a terminal state of the automata is reached. Semantic rules may also use auxiliary variables in order to store partial results during the automata evolution. The framework defines three types of properties: single values throughout the execution (such as, for example, the total system utilization), sequences of values (such as the response times of a task) or a statistically accumulated figure over a value sequence (such as the worst-case computation time of a task).

Overall, the framework establishes a direct relationship between properties, events and states. This relationship can be looked at from two alternative viewpoints, depending on the instrumentation possibilities in the target system:

a) the properties to extract conditions the states that the model system has to keep track of, which in turn defines which basic events *need* to be observed from the target system; or
b) the events that *can* be observed in the target system restricts which states can be derived in the system model and thus which properties can be extracted.

## 4 Study of Basic Temporal Properties

This section explains how to apply the framework described above in order to extract some relevant temporal properties of a generic (but typical) real-time system. The study presented here is independent of a particular application, as long as it follows the computational model presented below. For space limitation reasons, this paper only shows how to extract four temporal properties; however, the study below can be seen as a template by which other properties can be defined.

The study begins with the presentation of the computational model, that is, the set of assumptions that the target system must follow. Then, it presents the automaton model, which establishes the characteristics and notation of the automata that will be used to deduce properties. The final subsection shows how to actually extract some interesting properties of the target system: a system-related property (utilization) and three task-related properties (computation time, blocking time and response time per task release). For each property, the computational cost of the automata processing is also included.

### 4.1 Computational Model of the Target System

The target system considered here is formed by a real-time application plus an operating system, both running on a single-processor computer. The following list summarizes the behavioral restrictions that must hold on the system:

- The application comprises a fixed set of $N$ tasks.
- Each application task is either periodic or sporadic.
- Each task is statically assigned a priority.
- The operating system scheduler applies a fixed-priority preemptive dispatching policy.
- Tasks may share resources. If they do, resource access is exclusively arbitrated by using the Immediate Priority Ceiling Protocol.
- The *minimal* set of basic events that the instrumentation is required to report is the following: (1) for each context switch, an event denoted as CSW has to report the identifier of the new running task and the timestamp; (2) each time a task changes its runnable state, the new state has to be reported along with the task identifier and the timestamp. Significant states are: "ready" (RDY) and "finished" (FNS), respectively indicating that the task is runnable and finished (that is, suspended waiting for its next release); and (3) for each task, an event denoted as PRI has to report the initial (static) priority of each task, along with its identifier, when it is created.
- The process in charge of extracting the temporal properties is reported all the events without loss. The issue of building property automata which are robust under event loss conditions is currently being studied.

### 4.2 Event and Automaton Model

This section specifies a model for events and automata which is compatible with the framework. This model will be used below to define the properties under study.

Each event generated by the instrumentation is a tuple $(Y, t, i, \varkappa)$, where:

- $Y \in \{\mathsf{CSW}, \mathsf{RDY}, \mathsf{FNS}, \mathsf{PRI}\}$, that is, $Y$ is the event type.
- $t$ is the event timestamp.
- $i$ is the task identifier (all the events here are to be task-related). Task identifiers are ordered inversely with task priorities, with task '0' being the highest and task 'N-1' the lowest. Task 'N' here denotes the 'idle task'.

– $\varkappa$ denotes additional, event-specific information. For example, in events of type PRI, this will stand for the task new priority.

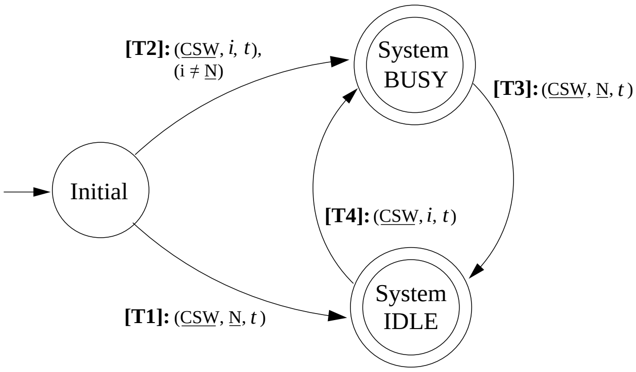
According to the framework, each property is conceptually specified by an automaton, which can be defined by five related elements: (1) a set of states, (2) a set of transitions, (3) a list of semantic rules attached to the states and/or transitions, (4) variables to store the property values, and (5) an optional list of auxiliary variables. Among them, the set of transitions is probably the only issue that needs further explanation, particularly about how a transition can express the conditions on which it becomes triggered.

In this automaton model, each transition is labelled with a condition of the form: "[**label**]**:** *pattern* [, *expression* [,. . . ]]". Inside this condition, the *label* simply identifies each transition. The *event pattern* is a tuple, equivalent to the event tuple, on which some of the elements are literal (constant) while some others can have a generic value. In the patterns of the automata below, the notation uses underline typeface for literals and italic typeface for generic values (i.e., values that can be instantiated). The optional *expressions* add more logical conditions to the transition to be triggered. In general, an event will trigger a particular transition if, first, the event matches the pattern (all the pattern literals occur in the event) and second, all the expressions is satisfied. Expressions are typically boolean conditions involving event data and/or some property values. The model also permits automata to have $\lambda$-transitions.

### 4.3 Some Property Examples

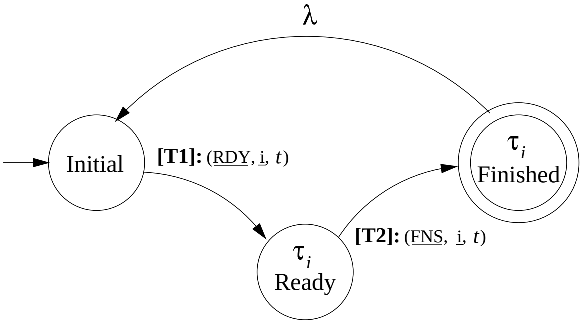
**System Utilization.** The system utilization can be easily calculated as 1 minus the utilization of the "idle" task. As a result, the only event needed to be observed for this property is the context switch (CSW). The utilization automaton, represented in Figure 2, needs two auxiliary variables, `itime` and `istart`, indicating the total accumulated idle time and the start of a idle period, respectively. The table in Figure 2 shows the semantic rules related to each transition. At any given time $t$, following formula can be used to calculate the system utilization: $U(t) = 1.0 - \frac{\mathsf{itime}}{t}$. The computational cost of this automaton is $O(1)$ with each event of type CSW.

**Response Time.** The response time of a task is calculated as the time on which the task finishes its release minus the last time it was released. In terms of events, this is calculated as the time elapsed between the pair of events RDY and FNS for any particular task. The automaton for calculating the response time of task $\tau_i$ is shown in Figure 3. Note that, due to the $\lambda$-transition, the automaton successively calculates all the response times of task $\tau_i$. The response-time automaton for each task $\tau_i$ needs two straightforward semantic rules. These rules operate with the property variable, called `rtime`, and an auxiliary variable to store the moment at which the release started, called `rstart`. The computational cost of this automaton is $O(1)$ for each RDY and FNS events corresponding to each task.




| Transition | Semantic rules |
|---|---|
| [**T1**] | itime := 0; istart := $t$; |
| [**T2**] | itime := 0; |
| [**T3**] | istart := $t$; |
| [**T4**] | itime := itime + ($t$ - istart); |

**Fig. 2.** System utilization automaton.

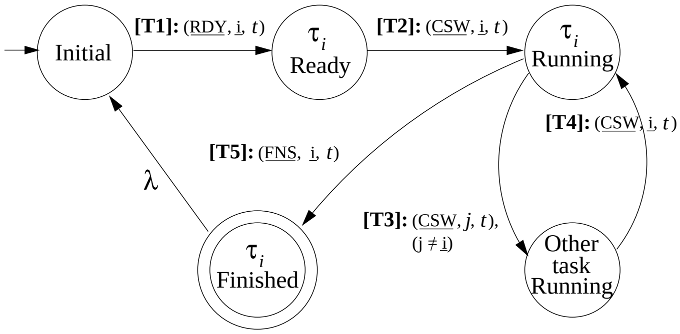



| Transition | Semantic rules |
|---|---|
| [**T1**] | rstart = $t$; |
| [**T2**] | rtime = $t$ - rstart; |

**Fig. 3.** Response-time automaton for task $\tau_i$.

**Computation Time.** The computation time of a task is the sum of the intervals on which the task is running. Since the computation time is a typical per-release property, the automaton has to calculate the task computation time of a single release; that is, between a RDY and a FNS event of a given task $\tau_i$. Apart from these two events, the automaton needs to get all CSW events that both put $\tau_i$

to run and remove $\tau_i$ from running. As in the previous section, a $\lambda$-transition is used to return to the initial state, in order to repeat the calculation release over release. The computation time automaton for task $\tau_i$ is shown in Figure 4. Semantic rules in Figure 4 use `ctime` as the property variable to calculate the task computation time over a release and the auxiliary variable `cstart` to remember the start time of a running interval for task $\tau_i$.




| Transition | Semantic rules |
|---|---|
| [T1] | ctime = 0; |
| [T2] | cstart = $t$; |
| [T3] | ctime = ctime + ($t$ - cstart); |
| [T4] | cstart = $t$; |
| [T5] | ctime = ctime + ($t$ - cstart); |

**Fig. 4.** Computation-time automaton for task $\tau_i$.

The computational cost of this automaton is $O(1)$ for events RDY and FNS and $O(2)$ for each event CSW, since this event effectively removes a task from running and puts another task to run (thus, the event causes a transition in the automata of both tasks).

**Blocking Time.** In the IPCP algorithm [Kle90], a task that wants to have exclusive access to a resource immediately raises its priority to the priority ceiling of that resource. In this protocol, a task $\tau_i$ may only be blocked when it is released, if a lower priority task $\tau_j$ has previously raised its priority to a ceiling which is higher than or equal to the priority of $\tau_i$. Once $\tau_i$ is chosen to run, no more blocking can occur to the task in this release. An example of this behavior is shown in Figure 5, which depicts a time diagram of a real application execution, featuring tasks, `A`, `B`, and `C` (in decreasing priority order). The figure shows the blocking of medium-priority task `B` by lower-priority task `C`, due to the fact that the latter has locked semaphore `m0` and raised its priority (at time 1) before the former is released (at time 2). The blocking lasts until task `C` unlocks `m0` and retrieves its original priority (at time 3). During the blocking interval, higher priority `A` preempts task `C`, but this execution interval is not to be considered blocking time for task `B`.

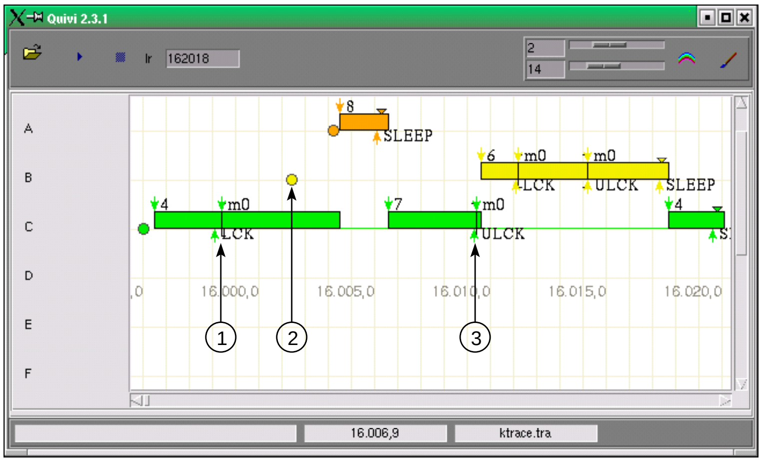
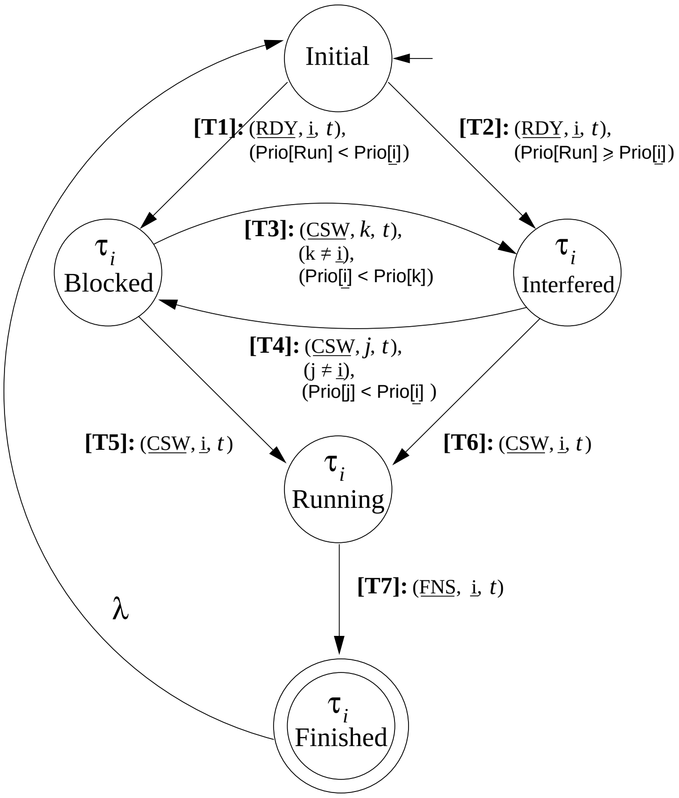



**Fig. 5.** Run-time scenario where task `C` blocks task `B`.

This behavior can be detected by an automaton that checks the priority of the running task when a given task $\tau_i$ is released. If the running task has lower priority, then blocking may happen, but only until $\tau_i$ gets to run. During this interval, the task which is blocking $\tau_i$ may be preempted by tasks with priorities higher than $\tau_i$, with these intervals not forming part of $\tau_i$'s blocking time. As explained for the computation-time automaton above, the blocking automaton also needs a $\lambda$-transition in order to successively calculate the blocking factor of each release of task $\tau_i$. The automaton is presented in Figure 6. Some transitions have logical tests in addition to event patterns, indicating that the transition is only produced if the pattern is instantiated by the current event and the condition is true at that moment. These conditions need two properties that have to be calculated by auxiliary, simple automata: the property variable Run stores the identifier of the currently running task; the property variable Prio[i] stores the initial, static priority of task $\tau_i$. The automata for these properties are straightforward.

The semantic rules for this automaton effectively accumulate the blocking time (`btime`) over a period of time between the task release and its first tick. In order to do so, an auxiliary variable (`bstart`) is used to store the start of a blocking interval. Note that this automaton could be simplified (for example, by removing the "$\tau_i$ Finished" state and making the "$\tau_i$ Running" to be the final state); however, we have chosen to present it in this form in order to illustrate




| Transition | Semantic rules |
|---|---|
| [**T1**] | btime = 0; bstart = $t$; |
| [**T2**] | btime = 0; |
| [**T3**] | btime = btime + ($t$ - bstart); |
| [**T4**] | bstart = $t$; |
| [**T5**] | btime = btime + ($t$ - bstart); |
| [**T6**] | (void) |
| [**T7**] | (void) |

**Fig. 6.** Blocking-time automaton for task $\tau_i$.

how release-like properties will typically use the FNS event type in order to mark the end of the property calculation.

The computational cost of this property is $O(1)$ for each RDY and FNS events and $O(N)$ for CSW events. A single context switch may affect the blocking state of many tasks at once. As a result, a CSW event may need to be given to the blocking-time automata of all tasks in the worst case.

## 5 Case Study

The previous section presented how to instantiate the framework to extract properties from a generic system. This section presents a case study in which the instantiation is followed a step further, presenting the low level, implementation issues that arise when these properties are extracted from a real system. In particular, the target system of the case study is a POSIX real-time application running on Open RT-Linux version 3.1, which has been enhanced by adding a POSIX Trace subsystem, as explained in Section 2.3. The section presents first the low-level system model, since it conditions the application behavior, and hence the event interpretation. Then, it explains how the target system has been instrumented in order to extract the required event types for analysis. And finally, it presents the *property extractor* process, which is the program that actually analyzes the trace and deduces the temporal properties.

The study presented in last section can directly be implemented in the case study because the target system follows the computational model presented above. In particular, sample applications have been implemented by using these restrictions:

- Each application consists of periodic and sporadic tasks with a static assignment of priorities. In particular, each task has been assigned a different priority.
- All tasks are scheduled according to the POSIX "SCHED FIFO" scheduling policy.
- Tasks sharing resources do so by means of using mutexes that follow the POSIX "PRIO PROTECT" protocol.
- Application tasks use a particular function to wait until its next release, with this function being *only* used for this purpose. In particular, periodic tasks use the absolute version of `clock_nanosleep` while sporadic tasks wait in a `pthread_cond_wait` call.

The mechanism used for extracting and retrieving events is the POSIX Trace system implemented in RT-Linux. According to the Trace standard, operating systems can introduce new *system* event types for their own needs. This possibility has been used here in order to extract as many of the required events as possible by instrumenting the RT-Linux kernel only. In fact, all the temporal properties studied so far (fully listed below) can be extracted by only analyzing a few system event types, which are compatible with the abstract event types defined in Section 4.1. The actual system event types that the RT-Linux kernel instrumentation produces are the following: (1) *context switch*, every time a new task is put to run, including the task identifier; (2) *task state change*, every time a task changes its runnable state, including both the task identifier and the new state; (3) *task priority change* when the task is created and each time that it explicitly changes its priority afterwards, including the task identifier and the new priority; and (4) *system call invocation*, every time a task invokes a system call of interest[4], including the invoked system call, the invoking task and any other relevant parameter in the call. According to the POSIX Trace standard, an automatic timestamp is registered for all the traced events, so this information does not have to be explicitly traced by the instrumentation.

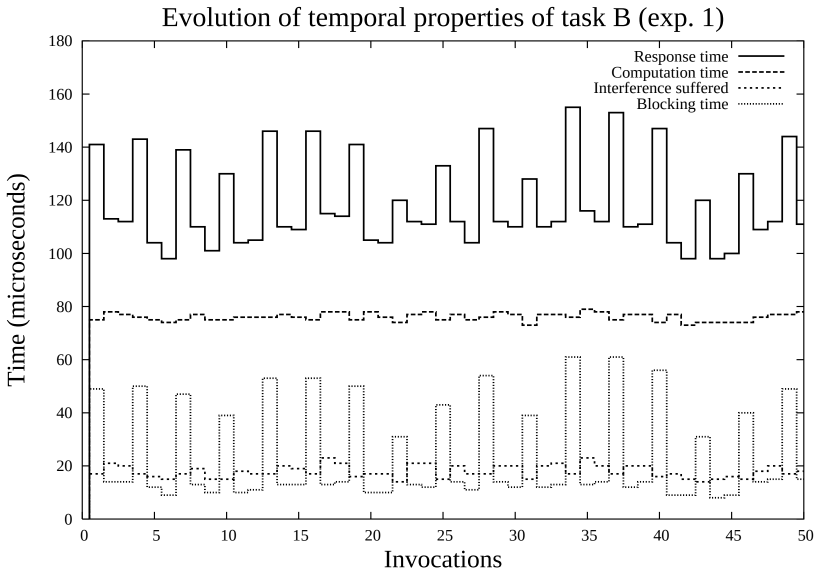
At run time, a stream without log is created in RT-Linux before the application begins to execute. This stream is then set to filter out both all user event types and each non-interesting system event type. Thus, the stream only registers the event types which are relevant to the property extractor process, effectively minimizing the tracing overhead. Early experiments have shown that the overhead of tracing and retrieving a single event is usually less than 500 nanoseconds (each) in a typical Pentium III processor (see [Ter01] for details).

The property extractor (PE) process (which corresponds to the framework's model system) has been implemented as a user Linux process. This program access the trace stream created in RT-Linux, retrieving all the events traced by the RT-Linux kernel as the real-time application runs. For each event retrieved, the PE process passes it to the appropriate automata, in order to calculate the temporal properties of interest. For each of these properties, a different automaton has been implemented as a as a table (of states versus events) containing pointers to the appropriate semantic rules. The PE main function contains the very simple and fast automata engine, which basically gets each new event, identifies which automata are "interested" in the event and then triggers the corresponding transitions in these automata. Our experiments show that the execution time of the PE typically takes less than $2\mu$s per automaton transition on a 850 Mhz. Pentium III processor. This low overhead makes possible the on-line analysis of a reasonable amount of properties, specially given the fact that, from the RT-Linux application viewpoint, the property calculation is done in the background (that is, by a Linux process).

Currently, the set of properties that the PE is able to calculate are: the system utilization, the current running task, the base priority (and successive priorities) of each task, and eight properties per task invocation: computation time, response time, blocking time, interference, inter-arrival time, release jitter, input jitter (begin-to-begin) and output jitter (end-to-end). For each task, the PE stores all the release-dependent values for a maximum number of task invocations, but also keeps some statistical values of each of them, such as the minimum, maximum and average.

Finally, just to illustrate the utility of such property extraction, let us consider the following experiment involving a very simple, three-task application which use a single *mutex* to achieve mutual exclusion over a resource. From highest to lowest priority, tasks are labelled `A`, `B` and `C` and their periods are 5, 10 and 15 ms, respectively. The experiment show the effect of blocking in task `B`, in two cases: (1) task `C` is released slightly earlier than the other two (80 $\mu$s); and (2) all tasks are released at the same time. Figure 7 shows the results of these experiments.

[4] For example, the system call that is known to signal the finish of the task current release.



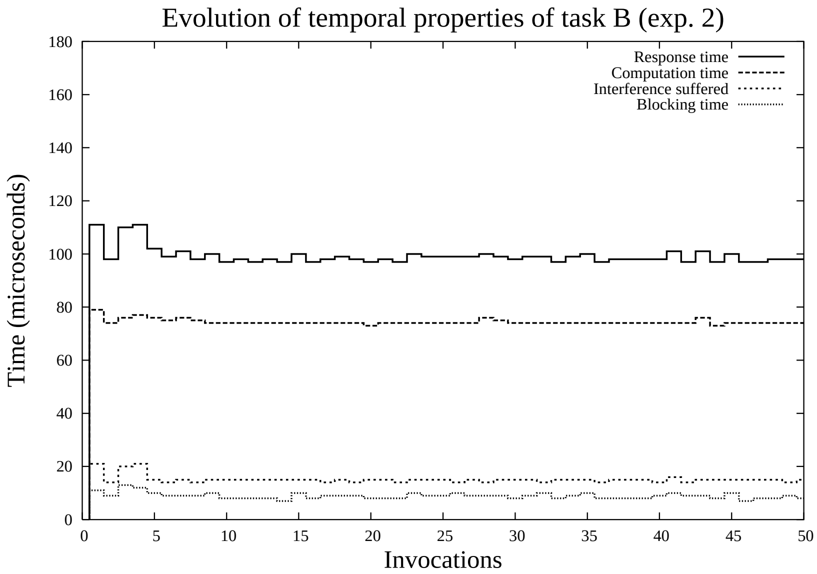




**Fig. 7.** Property extractor results for task `B` in experiments 1 (above) and 2. Experiment 1 had tasks `A` and `B` with a initial offset of 80 $\mu$s, while experiment 2 had all three tasks initially released at the same time.

The two graphics show the evolution of four temporal parameters of task `B` during its first 50 invocations, including the computation time, the blocking time, the interference and the response time. As it can be easily seen by comparing both graphics, the effect of the blocking time of task `B` practically disappears just by putting a different offset on task `C`, greatly decreasing task `B`'s response time. However, the off-line analysis would have calculated the same worst-case response time for task `B` in either case.

This small example shows the utility of the property extractor has for the designer, who can use its results to compare them with the analytical test. Furthermore, just by changing the configuration and run the system again, the exact effects that this change produces in the system behavior can easily be checked, in real conditions and independently of how big or complex the application is.

## 6 Conclusions

The main conclusion of this paper is that the application of a consistent framework for observing real-time systems can greatly enhance our knowledge about their behavior, compared to the results of the off-line analysis, which can only obtain the worst-case behavior. This paper has presented such framework, and has illustrated its capabilities by showing how some interesting temporal properties can be deduced from a generic but typical real-time system. Many other properties can be extracted by applying the same scheme. The constant (or, at most, linear) computational complexity of the property extraction process permits this process to run as a part of the real-time application, if necessary.

The type of temporal properties which are naturally interesting to know about a real-time system (computation times, response times, etc.) influences the fact that the instrumentation is mainly done at the operating system level. In this sense, the paper's case study has shown the ability of the POSIX Trace standard to provide such instrumentation. The conclusion of the case study is that POSIX-conforming real-time operating systems with the Trace option can be used to automatically extract temporal properties of applications by tracing a reasonably small set of system events. This, in turn, has proven the framework to be adopted by systems using current technology.

Currently, research is being done towards using the temporal property extraction process to build flexible schedulers that work with actual observations rather than off-line estimations. Other research lines about this subject include fault-tolerant schemes that can deal with event loss in the process and a full characterization of the relationship between the event types that can be observed from a system and the temporal properties that can be extracted.

## References

[P1003.13] "1003.13-1998 IEEE Standard for Information Technology–Standardized Application Environment Profile (AEP)—POSIX® Realtime Application Support" [0-7381-0178-8].

[P1003.1] "1003.1$^{TM}$ Standard for Information Technology—Portable Operating System Interface (POSIX®)". IEEE Std. 1003.1-2001, Open Group Technical Standard Base Specifications, Issue 6.

[Alur94] Alur R., and Dill, D.L. (1994). "A Theory of Timed Automata". Theoretical Computer Science Vol. 126, No. 2, April 1994, pp. 183-236.

[Aug95] Auguston, M. (1995). "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", Proc. of the 2nd Intl. Workshop on Automated and Algorithmic Debugging, Saint-Malo, France, May 1995.

[Bar77] Bartussek, A.W., and Parnas, D.L. (1977). "Using traces to write abstract specifications for software modules". UNC Rep. TR 77-012, Univ. North Carolina, Chapel Hill.

[Bat95] Bates, P. (1995). "Debugging heterogeneous distributed systems using event-based models of behavior". ACM TransactIons on Computer Systems, Vol 13, No 1, Feb. 1995, pp. 1–31.

[Bro84] Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W. (1984). "A Theory of Communicating Sequential Processes". Journal of the ACM, Vol. 31, No. 3, July 1984, pp. 560–599.

[Kle90] Klein, Mark H. and Ralya, T. (1990). "An analysis of input/output paradigms for real-time systems". Technical Report, Software Engineering Institute. CMU/SEI-90-TR-19, 1990.

[McL84] McLean, J. (1984). "A formal method for the abstract specification of software". Journal of the ACM, Vol. 31, No. 3, July 1984, pp. 600–627.

[Ste92] Stewart, D.B., Schmitz, D.E., and Khosla, P.K. (1992). "The Chimera II real-time operating system for advanced sensor-based control applications". IEEE Transactions on Systems, Man, and Cybernetics, Vol. 22, No. 6, Nov./Dec. 1992, pp. 1282-1295.

[Ter01] Terrasa, A., Pachés, I., and Gacría-Fornes, A. (2001). "An Evaluation of the POSIX Trace standard implemented in RT-Linux". Proc. of the 2001 IEEE Intl. Symposium on Performance Analysis of Systems and Software, Tucson (AZ), pp. 30–37.

[Wan93] Wang, Y., and Parnas, D.L. (1993). "Simulating the behaviour of software modules by trace rewriting". Proc. of the 15th intl. conference on Software Engineering, Baltimore (MA), May 1993, pp. 14–23.

[Yod99] Yodaiken, V. (1999). "An RT-Linux Manifesto". Proc. of the 5th Linux Expo, Raleigh, North Carolina, May 1999.